PrefixStream: A Balanced, Resilient and Incentive Peer-to-Peer Multicast Algorithm

Anh-Tuan Gai, Laurent Viennot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

PrefixStream: A Balanced, Resilient and Incentive Peer-to-Peer Multicast Algorithm

Anh-Tuan Gai — Laurent Viennot

N° 5514

Mars 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5514--FR+ENG

PrefixStream: A Balanced, Resilient and Incentive Peer-to-Peer Multicast Algorithm

Anh-Tuan Gai*† , Laurent Viennot*†

Thème COM — Systèmes communicants
Projet Gyroweb

Rapport de recherche n° 5514 — Mars 2005 — 19 pages

Abstract: We consider the problem of multicasting a stream of packets in a large scale peer-to-peer environment. In that context, we stress three features: forwarding load should be equally balanced among nodes, the scheme should be resilient to node failures and peers should have incentive to cooperate. Mainly based on the seminal work of SplitStream which partially achieves this goals, we propose an algorithm gathering together these three features. Its main advantage is to reduce the forwarding load of every node to the stream bandwidth (every node uploads as much as it downloads). This ultimate load balancing is achieved together with a clustering scheme allowing bi-directional exchanges. This results in resilience to node failures and the possibility of banishing nodes that do not respect reciprocity of exchanges. This paper promotes disjoint clustering as opposed to previously proposed hierarchical clustering schemes. Interestingly, varying the size of clusters allows to obtain different trade-offs between delay optimization and resilience to node failures. The performances of several algorithms are analyzed and compared with respect to these goals. The propagation delays of these algorithms appear to be within a factor 1.5 to 2 from theoretical optimal.

Key-words: peer-to-peer, streaming, multicast

* INRIA Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay, France, {Anh-Tuan.Gai, Laurent.Viennot}@inria.fr

† Travail financé par le projet PairAPair de l'ACI Masse de données

PrefixStream: un algorithme de diffusion pair à pair équilibré et résistant aux pannes qui incite à la coopération

Résumé : Dans ce papier, nous considérons le problème de la diffusion d'un flux de paquets dans un environnement pair à pair de grande échelle. Dans ce contexte, nous cherchons à obtenir trois propriétés: la charge lié à la diffusion doit être répartie équitablement sur l'ensemble des pairs, le schéma de diffusion doit être résistant face à la volatilité des noeuds et les pairs doivent être incités à coopérer. En partant du travail séminal de SplitStream qui en possède déjà une partie, nous proposons un algorithme qui possède l'ensemble des propriétés souhaitées. Son principal avantage est de réduire la charge de chaque noeud au débit du flux (chaque noeud retransmet autant de données qu'il en reçoit). Cette répartition idéale de la charge est atteinte par un processus de regroupement des noeuds permettant également des échanges bidirectionnels. Il en résulte une résistance aux pannes et la possibilité de bannir les noeuds ne respectant pas la réciprocité des échanges. Ce papier prouve également l'intérêt de construire des groupes noeud-disjoints par opposition aux processus de regroupement hiérarchiques proposés dans de précédents papiers. En faisant varier la taille des groupes, on obtient un intéressant compromis entre l'optimisation du délai et la résistance aux pannes. Les performances de différents algorithmes sont analysées et comparées en fonction des propriétés recherchées. Le délai de propagation de l'ensemble de ces algorithmes est à un facteur qui varie entre 1,5 et 2 de l'optimal théorique.

Mots-clés : pair-à-pair, diffusion

1 Introduction

This paper proposes an algorithm for multicast streaming in a very large scale peer-to-peer network. We call multicast streaming an application where a source is sending a flow of packets to a large number of receivers. IP multicasting is certainly the most efficient way for multicast, however the burden of duplicating packets is carried by intermediate routers which are often independent from the source and the receivers. This may explain why transit networks hardly ever implement IP multicasting.

On the other hand, attention is now centered on end-system or application-level multicast where the participants duplicate themselves the packets [4, 3, 20, 10, 16]. Most of the existing algorithms still duplicate packets along a tree. However, a forwarding node with d sons will have to send every packet d times requiring an upload bandwidth d times greater than the stream bandwidth. This brings a major difficulty when forwarders may be end user receivers having a very low upload capacity. SplitStream [4] solves this problem by using d interior-node-disjoint trees such that every node is a forwarder in one tree and a leaf node in others. This scheme allows to multicast a stream whose bandwidth is close to the minimal upload capacity of nodes. SplitStream is designed to adapt to variable node capacities. Different forwarders may thus have different degrees. For this reason some extra-upload is needed to give sufficient freedom to the system when nodes continuously enter and leave the network. The algorithm can thus only guaranty that nodes will use an upload roughly equal to the stream bandwidth. Moreover, reconstruction after a node failure can be rather intricate.

A second difficulty resides in the reliability of nodes. If a forwarder fails, it has to be replaced very quickly to re-establish the feeding of downstream nodes. With a large number of participants, this event could happen quite frequently. To make reconstruction very fast, Nice [3] and Zigzag [20] uses a similar hierarchy of clusters to build a single multicast tree. When a node fails, it can be replaced by some other node of its cluster. Clustering thus gives these schemes good resilience to node failures, but the presence of powerful forwarders with upload bandwidth 4 to 10 times higher than the stream bandwidth is still assumed.

Design goals: We target a large scale system where forwarding is made by end users themselves. (Think of a radio stream received and forwarded by one million ADSL users for example.) In that context many nodes may have a limited upload capacity. Every node is thus required to have a minimal upload capacity u to participate in the network. With the stream bandwidth normalized to 1, u should ideally be as close to 1 as possible. (Indeed, we target an algorithm with upload bandwidth equal to 1 for every node.)

Moreover, we suppose that nodes need incentive to cooperate: we suspect that a node with greater capacity will not intend to spend more than an upload u when it can enter the network and receive the same stream quality by spending only u. Such behavior has been observed in file sharing applications where free riders [2, 18] tend to upload only if they cannot download otherwise. BitTorrent [5] made a major breakthrough in that domain by building incentive through tit for tat.

This selfish behavior thus induces the assumption that every node will approximately give the same upload u to the application. In that sense we are looking for a balanced algorithm. A broader scenario would consider nodes with varying upload capacities where nodes spending more capacity intend to receive a better stream quality. A pretty simple way to achieve this within our framework is to set a different network for each upload capacity. (Looking at typical Internet connections, users can easily be grouped under three or four different upload capacities.) Considering the video streaming case, multiple description coding [9] can even allow to gather the streams broadcasted in each network consistently. (Nodes with more upload capacity then participates in more networks.) The balanced scenario is thus a key point in order to solve this broader scenario. For that reason, the paper focuses on the balanced case.

Resilience is also a major goal concerning large scale peer-to-peer systems since many nodes may simply fail and leave the network without any previous announcement. Finally the maximal delay is a classical concern in multicasting especially for live streaming.

Contribution: The main idea of this paper is to combine both approaches of SplitStream and Nice/Zigzag to achieve both goals of resilience and forwarding-load balancing. We propose an algorithm called *PrefixStream* that combines both approaches with their respective benefits. The main advantage of our algorithm is that all nodes use an upload bandwidth exactly equal to the stream bandwidth and this feature is still preserved in case of node departures and arrivals. The idea of combining both approaches is simple but has required to design completely new schemes for node clustering and disjoint tree construction to achieve this goal. Basically our algorithm forms clusters by gathering together all the nodes whose ID share a given prefix. We then construct d interior-cluster-disjoint trees. The choice of gathering by prefix is natural for allowing an efficient distributed clustering algorithm. However, it imposes a new tree construction algorithm compared to SplitStream.

Moreover, our algorithm solves the problem of giving incentive to cooperate, an inherent difficulty of completely decentralized peer-to-peer systems. Inside a cluster, a non-forwarder can be detected and banished by other members. Links between clusters can be used in both directions (one tree will use it in one direction and another tree will use it in the other direction). A node omitting to forward inter-cluster traffic can be detected and banished by the other cluster nodes.

Network Model: The number of nodes will be denoted by n. The source is an external node reliably furnishing the packets with a bandwidth 1 (the source do not duplicate packets). As we suppose that all nodes give approximately the same upload bandwidth for the multicasting application, we will simply suppose that all nodes have the same upload capacity $u \geq 1$. To analyze delays, we use the following model of transmission: a packet requires a time T to be emitted and transits a time L in the network. (When sending i packets in a row, the last packet will thus be received after a time $iT + L$.) With a 128 kbit/s upload bandwidth and 1000 bits packets, T is typically less than 10 ms. L is half of the average round trip time on the Internet and varies typically between 20 and 150 ms. This is similar to the LogP model [6] except that we suppose a fixed packet length. Lower bounds on the

Algorithm	*Reconstruction overhead*	*Maximal upload*	*Worst failure packet loss*	*Maximal delay*	*Buffer overhead*
Single Tree	$O(hd)$	d	1	$h(d+l)$	0
SplitStream [4]	$O(hd)$	1	$1/d$	$h(d+l)$	1
Zigzag [20]	$O(m)$	$(m-1)^2$	1	$h((m-1)^2+l)$	0
Single Cluster	1	$(n-1)/n$	$1/n$	$(n-1)+l$	$n-1$
Cluster Tree	m	$1+(d-1)/m$	$1/m$	$(h-1)(d+l)+m+l$	$m-1$
PrefixStream	d	1	1/2d	$h(d+l)+m+2l$	$m-1$

Table 1: Analysis of the various peer-to-peer streaming algorithms considered in the paper. d designates the degree of forwarding nodes. $h = \lceil \log_d n \rceil$ is the height of a tree of degree d with n nodes. m is the cluster size. $l = L/T$ is the normalized latency in time for emitting a packet. All values are normalized either in number of packets or messages or in time for emitting a packet.

minimum delay for multicasting a packet to n nodes can be found in [12]. The normalized latency $l = L/T$ is the maximal number of packets a node can send in a row before the first packet is received. ($d = l$ gives some rough estimation of the optimal degree for forwarders.) A simple theoretical lower bound on the normalized delay is hl [12]. We will see that a tree of degree d allows to achieve a delay $2hd$ which is at most twice longer than optimal for $d = l$.

We use this simple model to analyze and compare the different algorithms for peer-to-peer multicasting. However, in practice, L may vary a lot, especially for a world wide application. Clustering allows to optimize delays with this regard. In our algorithm, a forwarder x sending a packet to a cluster has the liberty to select the node y to reach in the cluster. It can thus choose the node with smallest round trip time.

As packets may arrive at receivers out of order, an important measure is the number of packet that need to be buffered in order to reorder the stream. This buffer indeed causes some additional delay to play the stream. We will just count the reordering due to the algorithm chosen for forwarding packets. The jitter due to variations of the transmission delay will not be included in the count as it is roughly the same for all algorithms. When entering the network, this buffer time induces a response time for the user: the time she has to wait between her decision to listen to the stream and the time when the stream begins to be played.

To evaluate the reliability of the different algorithms, we compute the worst failure packet loss. When a node fails, we can define the average packet loss as the average fraction of packets lost by remaining nodes. This packet loss can greatly depend on which node fails. The worst failure packet loss is defined as the packet loss obtained in the worst case (i.e. the most important node fails, typically the root of a tree). This measure is important because it expresses how a single node failure may impact the majority of nodes. Notice that a few random failures in a tree have less impact than the failure of the root. We also discuss the large number of failures case (for example during a partial network breakdown) with regard

to the ability to recover such a dramatic scenario. Finally, we define the tree reconstruction overhead as the time needed to rearrange the topology of the system when a node fails and has to be replaced. We will estimate it by counting how many messages must be sent during reconstruction. Again we consider the worst case where the most unfavorable node fails. This measure estimates the resilience of the scheme to a node failure.

Table 1 gives a summary of the analysis of the peer-to-peer streaming algorithms considered in the paper. To get a consistent view, the algorithms are supposed to use the same degree d for forwarders. The multicast tree constructed thus will have height $h = \lceil \log_d n \rceil$. This table basically shows that PrefixStream allows to have fast reconstruction time and balanced upload at the cost of slightly more delay than a regular tree of degree d. (The delay bound of PrefixStream is approximately twice the theoretical optimal delay when SplitStream is within a factor 1.5 from optimal.)

The basic tree multicasting algorithm uses a tree of degree d and height h. SplitStream uses d trees of degree d. Zigzag uses clusters of size $m = \Theta(k)$ (k is a parameter of the algorithm) and a tree of degree $d = m - 1$ and height $\log_k n \approx \log_d n$. However some nodes may have degree $\Omega(k^2)$ requiring an upload $(m-1)^2$. The single cluster is a study case algorithm that uses in turn n trees of height 1. The cluster tree consists in a tree of clusters of size $m = d$ (the tree height is thus $\log_d(n/d) = h - 1$). PrefixStream uses 2 sets of d interior-cluster-disjoint cluster trees.

2 Background

2.1 Single Tree Architecture

The basic architecture for broadcasting consists in organizing the receiving nodes in a *tree* of degree d. The source sends the stream to the root of the tree. Every interior node receives each packet from its parent and forwards it to its d sons. Leaf nodes simply receive packets without forwarding.

First notice that this scheme is not balanced in our sense since the leaf nodes do not forward while interior nodes need an upload bandwidth d. As the packets arriving to a receiver all travel along the same path, they are received in the emission order. Let $h = \log_d n$ be the tree height. The broadcast delay is bounded by $h(dT + L)$ since a node of the tree receives a packet at most $dT + L$ after its parent. To minimize jitter, the emission towards the sons should remain the same. (The forwarders of any algorithm should keep this as a rule of thumb.) If we normalized to $T = 1$, the maximal delay is thus $\frac{\log n}{\log d}(d + l)$ which is minimial for $\log d - \frac{d+l}{d} = 0$. Figure 1 illustrates the degree d giving optimal delay as a function of l. This curve shows that typical values of l (from 2 to 15) require a degree from 4 to 11.

Figure 2 illustrates the maximal delay obtained with the tree architecture for various values of d. As a hint of comparison, we have computed the theoretical optimal delay for multicasting one packet to $n = 10^6$ nodes. The best theoretical scheme consists in all nodes

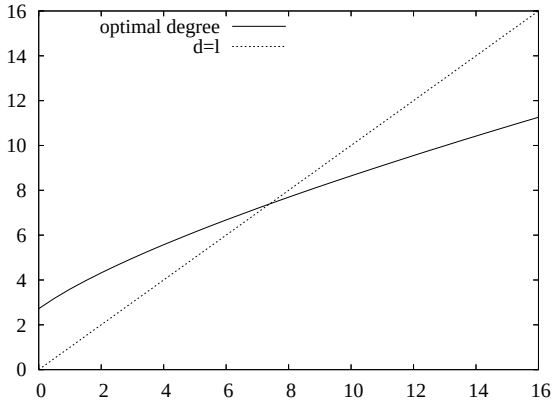

Figure 1: The optimal degree d of a multicast tree as a function of $l = L/T$, the normalized latency.

forwarding continuously the packet as soon as they receive it and as long as some node has still not received it [12] (every emission is supposed to reach a new node). It can thus be defined as the smallest index i such that $a_i > n$ where a_i is the sequence defined by $a_i = a_{i-1} + a_{i-l-1}$ and $a_0 = 1$ and $a_{-1} = \cdots = a_{-l} = 0$ (see [12] for the details). For $2 \leq l \leq 16$ the tree delay is roughly 1.5 times the optimal delay for any degree $4 \leq d \leq 8$. This shows that a fixed degree can give good delays for a wide range of l values. This is a good point when the parameter d has to be fixed once for all before deploying a multicasting network.

We now analyze the reliability of the scheme. First consider the node failure worst case: if the root fails all nodes loose all packets until the tree is repaired. This yields a worst failure packet loss of 1. Notice that packet loss due to sporadic lack of forwarding is simply proportional to the average number of forwarders for reaching a node (this is true for any scheme). The worst case for reconstruction time again occurs when the root fails. We can then show a hd lower bound on the number of messages exchanged to repair the tree by assuming an iterative reparation process: the root is replaced by one of its sons ($d + 1$ messages are then necessary to inform the sons of the son and the source), the new root is then replaced by one of its sons and so on along a path from the root. A faster scheme could consist in directly selecting a bottom node for replacing the root. However, this would imply additional topology maintenance. Doing this without any conflict when several nodes fail is not obvious. An algorithm for finding nodes with spare capacity would then be a possible solution. SplitStream uses such a mechanism with again a $O(hd)$ bound (see Section 2.3).

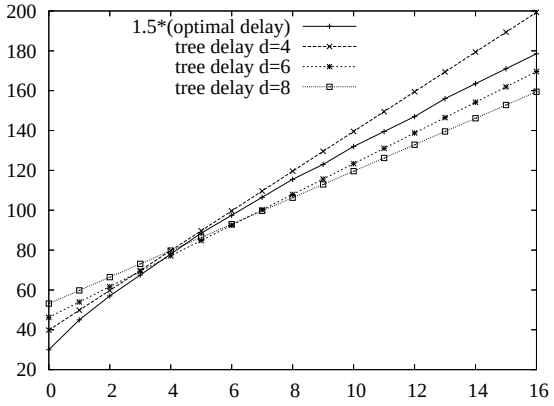

Figure 2: The maximal delay obtained with the tree architecture as a function of d.

2.2 Cluster Architecture

We call *cluster* a group of nodes knowing each other. A simple scheme for broadcasting a packet in a cluster consists in giving each packet to one node which then forwards it directly to all other nodes. The source can choose each node in turn for accomplishing this task.

Notice that this scheme is unfeasible for a large number of nodes. However, it is balanced in our sense. All nodes forward one packet over n to the $n-1$ other nodes, requiring an upload of $\frac{n-1}{n} \leq 1$. The delay is bounded by $(n-1)T + L$. The packets may have to be reordered through a window of $n-1$ packets.

The advantage of this scheme is its reliability: all node failures being symmetric, the worst case for a node failure can only affect one packet over n until the detection of the node death. The worst failure packet loss is thus $\frac{1}{n}$. Moreover, if a node leaves the network, the reconstruction overhead simply consists in informing the source. Informing other nodes is less urgent. The source can piggy-back the information in packets it serves later on.

2.3 SplitStream

Splitstream [4] is based on a set of d interior-node-disjoint trees. Each node belongs to one tree as an interior node and in the $d-1$ others as a leaf node. Figure 3 illustrates an example of SplitStream architecture. The basic approach consists in using trees of degree d. However, a scheme is proposed to adapt the degree of each node to its upload capacity. In our framework where all nodes have same upload capacity, SplitStream would use regular trees of degree close to d.

This scheme is balanced since each node forwards one packet over d to its d sons in the tree where it is an interior node. All trees having height $h = \log_d n$, the delay is bounded

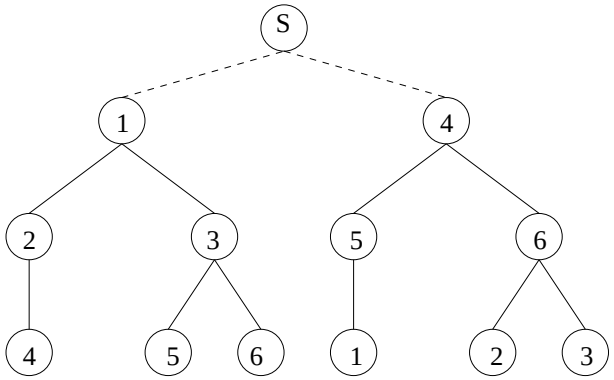

Figure 3: SplitStream (two interior-node-disjoint trees: the leaves of one tree are interior nodes of the others).

by $h(dT + L)$. With the SplitStream scheme, one packet over d arrives with a shorter delay: the packet received by the tree where the node is interior. To reorder packets, a node may need the additional buffering of one packet.

Concerning reliability, the authors argue that SplitStream can greatly benefit from techniques splitting the stream into d sub-streams (called stripes) such that loosing one stripe has a minor impact .(They propose for example to use multiple description coding for video and erasure coding for reliable distribution of data.) Any architecture based on several trees may benefit from such scheme. We just analyze the bare bone algorithm. The worst case for node failure occurs when one of the d roots fails. This gives a worst failure packet loss of $1/d$ as all nodes will then receive all stripes but one. To repair a tree, all the sons of the failing node must re-insert themselves in the tree. Each of the d sons must thus perform a Pastry lookup which requires approximately h messages. Each one can find a new parent this way. However, the process may be more intricate when the new parent has exhausted its upload capacity (if it has already d sons). A "push-down" process can then be iteratively performed to reject one child one layer deeper in the tree. This process can again generate $O(h)$ messages. Some child may then become orphan and have to find a parent in the spare capacity group. All the nodes with spare upload capacity are member of a special tree for that purpose. Any orphan thus sends an anycast message to the spare capacity group by performing a DFS like search in the tree until a node acting as an interior node for the right stripe is found. The complexity of this anycast scheme is hard to estimate. However, this group either very small (if all nodes have degree d) or a node will probably be found in $O(h)$, the height of the tree if the group is large. Cases with an $\Omega(n)$ search seem very unlikely. The overall complexity of reparation in case of one node failure is thus $O(hd)$.

2.4 Zigzag

Nice [3] and Zigzag [20, 19] use a similar hierarchical clustering scheme. We have chosen to discuss Zigzag which includes better description and analysis of its multicast tree construction. Zigzag uses a layered multicast tree where nodes are arranged into a hierarchy of clusters of size $m \in [k, 3k]$. A node belongs to at most one cluster at any layer and all nodes belongs to layer 0. If a node highest layer is i, it also belongs to every layer from 0 to i and it is a cluster head in layers from 0 to $i-1$. Heads inherit all the administrative functions except the responsibility of forwarding the data to other nodes. Non-head nodes of a cluster at layer i forward packets to all the non-head members of some of the clusters at layer $i-1$. (Head-nodes of clusters at layer $i-1$ belong to a cluster at layer i and thus receive the packets from a higher level.) Ideally, each node forwards to a constant number of clusters, yielding a $O(k)$degree. However some nodes may happen to have a degree $d = O(k^2)$. Figure 4 illustrates Zigzag's architecture.

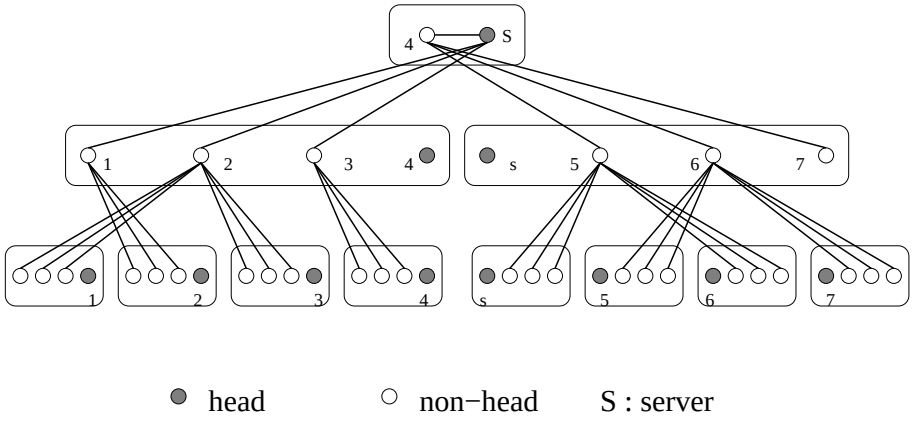

Figure 4: Zigzag's Architecture

Zizag performances are similar to a tree of degree $d = m - 1 = O(k)$. As the maximal degree is $(m-1)^2$, this delay is only bounded by $h((m-1)^2T + L)$ but the average expected delay is $h((m-1)T + L)$. The maximal required upload is $(m-1)^2$ and the protocol is not balanced in our sense. If the root fails, the packet loss is again 1. However, the reconstruction time of Zigzag is very short. The authors of Zigzag claim a $O(k^2)$ reconstruction overhead [20], the main term is due to the re-connection of the d sons of the failing node. However, a better bound is obtained when counting only the number of messages required for reconstruction. When a node x fails, the cluster head of each cluster whose non-head members where receiving the stream from x is responsible for finding a new parent. This is possible because x should only forward to clusters it does not belong to according to Zigzag architecture constraints. This basically requires $2(m-1)$ messages since x was feeding at most $m-1$ clusters. (One message per cluster to inform the head plus one message to inform

the new parent per cluster.) A new cluster head for all the clusters of x is designated in the lowest level cluster of x. If a cluster becomes too small, it is merged with the smallest cluster of the same layer. This again requires $O(m)$ messages but it does not impact reconstruction time since it can be done after reparation. This should rather be considered as topology maintenance.

3 PrefixStream

We first introduce a basic cluster tree scheme. This architecture is a simple novel multicast architecture inherently interesting. It is also a good introduction to PrefixStream which can be viewed as a union of cluster trees. Clusters and tree links are formed through the use of a distributed hashtable.

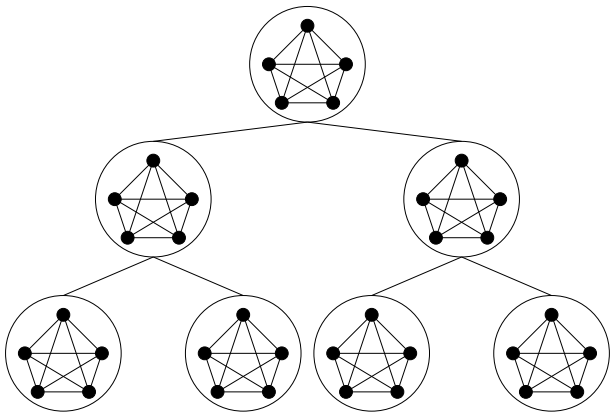

Figure 5: A cluster tree with $d = 2$ and $m = 5$

3.1 Cluster tree

We call *cluster tree* a tree of degree d where nodes are replaced by disjoint clusters of m peers. A peer knows all the members of its cluster, its parent cluster and its son clusters. A node thus knows $m(d+2)$ contacts but nodes forward only to $d+m-1$ nodes a given packet. When a node receives a packet directly from a node of its parent cluster, it forwards it to one node per son cluster and then to all other members of its cluster. Figure 5 illustrates the cluster tree architecture. To balance the load of forwarding, each peer of a cluster is selected in turn as a forwarder. A simple way to achieve this is to send the ith packet to the node in position i modulo m in the cluster list of nodes.

Cluster tree performances are similar to regular trees performances when $m = d + 1$. First notice that its height is $\log_d(n/m) = h - 1$. The maximal delay is thus bounded by

the time for the tree propagation $(h-1)(dT+L)$ plus the intra-cluster forwarding time $(m-1)T+L$. For $m=d+1$, we get the same delay bound as for a regular tree of degree d. The drawback of our way of clustering is that a window of $m-1$ packets is needed to reorder packets obtained inside the cluster. When u is close to 1 this can be viewed as an additional delay of $(m-1)T$. This also increases the response time when entering the network by the same value. However, for a small m like $m=2L/T+2$ this delay equals $2L+2T$ which is the time required to contact the network entry point and get its answer. This additional delay is thus affordable for $m=O(d)$.

Now consider the forwarding load of nodes. Every node forwards one packet over m to $d+m-1$ other nodes. This scheme is thus balanced and requires a maximal upload of $1+(d-1)/m$. Interestingly it is only 1.5 for $d=m=2$ and it is always less than 2 when $m \geq d-1$. Notice that this bound is valid independently from the size of other clusters. Finally, consider a node failure. The worst case occurs when a node of the root cluster fails. All nodes then loose one packet over m. The worst failure packet loss is thus $1/m$. The reconstruction time requires m messages: all the nodes of the parent cluster have to update their list of members for the cluster where a node has failed. Informing other nodes of the cluster is less important since it only affects unnecessary forwarding to the dead node. Moreover, the nodes in the parent cluster can piggy back the information when they forward a packet to some other member of the cluster.

If an efficient algorithm for maintaining the clusters is given, this cluster tree algorithm already seems an attractive solution for balanced and resilient peer-to-peer multicasting. This is the subject of the next section.

3.2 Forming clusters

We propose to form clusters by gathering together all nodes with IDs sharing a common prefix of length $p \approx \log_2(n/m)$. (This new idea is more in the spirit of Kademlia [14] than of Nice [3] or Zigzag [20].) A simple way to achieve this is to give random IDs to nodes and to define p as the length of the shortest prefix shared by less than m_0 nodes minus one. However, some clusters may then be bigger than others by a factor of $O(\log n)$. We will give later on hints for getting more equally balanced clusters.

First consider the issue of organizing the clusters in a tree. Notice that our clusters can be numbered by the prefix of p bits defining them. An elementary way of constructing a complete binary tree on integers ranging from 1 to 2^p is the heap structure where each node i has sons $2i$ and $2i+1$ (if they exist) and parent $i/2$. Sons are thus obtained by shifting the bits of i to the left and by inserting a 0 or a 1 at the lowest bit position. The parent is obtained by shifting the bits of i to the right and by inserting a 0 at the highest bit position. The tree structure of a heap is thus a sub-structure of the de Bruijn graph (which is defined by these shifting operations). In that sense, we could consider 1 as the unique son of 0, obtained by shifting 0 to the left and inserting a 1.

A general way to obtain a tree of degree d consists in using IDs which are members written in base d. Consider all the p digits numbers. A left tree is obtained by shifting

digits one position to the left: the d sons of i are obtained by shifting i one digit to the left and inserting a new digit on the right. Notice that a number where all the digits are the same will have only $d-1$ sons. Similarly, right trees are defined by shifting to the right and inserting a new digit on the left.

Each choice of a root gives a different left tree and a different right tree but all these trees share many edges. Consider a left tree with root $x \cdots x$ with all digits equal to x. Its sons will have a prefix of the form $x \cdots xy$, its grand sons will have prefix of the form $x \cdots xyz$ and so on. Notice that all interior nodes have x as first digit. Using different x thus yield interior-node-disjoint left trees. Interior-node-disjoint right trees can be obtained similarly. Trying to arrange our clusters in a tree of degree d, we have indeed constructed two sets of d interior-cluster-disjoint cluster trees.

Note that the de Bruijn graph has already been suggested for constructing peer-to-peer distributed hashtables [15, 7, 11, 1]. These four propositions were simultaneously discovered [7]. A later proposition [8] follows these tracks with the Kademlia [14] approach for resilience to node failures. The construction of cluster trees as described here follows the same ideas around the de Bruijn graph but it is independent from the overlay network used. A classical prefix routing distributed hashtable such as Pastry [17] or Kademlia [14] can also be used to compute clusters. Indeed, we can use any overlay network enabling efficient computation of the list of nodes whose ID begin with a given prefix. Kademlia seems a good candidate since it allows to find the k closest nodes to some ID with one lookup where k is a parameter of the protocol. Each node can then learn all the necessary contacts to construct a left tree with $d+2$ lookups: one for its cluster, one for its parent cluster and d for its son clusters. $d-1$ more lookups allow a node to learn its remaining right contacts for constructing any right tree or any left tree.

As mentioned before, getting equally balanced clusters is important with regard to delay performances. The ID assignment scheme should thus try to maintain a uniform allocation of IDs. A rather classical way to achieve this is to make new nodes choose $\log n$ random IDs and select the ID falling in the smallest cluster. One can then expect cluster sizes to be within a $O(1)$ factor (see [13]).

G. Manku proposes a smart way to choose node IDs and to re-assign at most one node ID in case of node departure [13]. This scheme can be adapted to any distributed hashtable overlay and reduces insertion and deletion overheads to the cost of one lookup approximately. This algorithm allocates IDs in a binary tree where all prefixes of length $[\log_2 n]+1$ are shared by at most one ID and where all prefixes of length $[\log_2 n]-1$ are shared by at least one ID with high probability. $[x]$ designates the integer closest to real number x. All prefixes of length $p = [\log_2 n] - 1 - \log_2 m_0$ will thus be shared by at least m_0 nodes and at most $4m_0$ nodes. If we impose that p is a multiple of $\log d$ (d is assumed to be a power of 2), this upper bound may have to be multiplied by $d/2$. For $m_0 = 2$, we get an upper bound of $m = 4d$ on cluster size. Interestingly a two ID re-assignments per node departure scheme [13] could be used to get clusters of size $d \leq m \leq 2d+2$.

3.3 PrefixStream architecture

The topology maintenance of PrefixStream requires a distributed hashtable overlay such as Kademlia [14] coupled with an ID assignment algorithm such as [13] in order to form clusters and to allow the members of the $2d+1$ clusters it may communicate with to be learned by a node. This part is only succinctly described here.

The basic PrefixStream architecture consists in using a set of d interior-cluster-disjoint cluster trees as defined above. We first consider the left trees obtained with roots $0\cdots0, \ldots, (d-1)\cdots(d-1)$. Each cluster is an interior cluster in the tree whose root shares the same first digit and a leaf cluster in other trees. The balancing algorithm of the cluster tree architecture seems difficult to extend here. This is due to the fact that d different clusters furnish packets to a given cluster and they would have to synchronize to avoid giving d packets to the same node in a row. For that reason, we propose to elect a cluster head per cluster.

The source selects one cluster head per root cluster. Each cluster being interior in only one cluster tree its cluster head is designated by the cluster head of its parent. The forwarding algorithm is now the following. The source sends packets to the d root cluster heads in turn. Each cluster head receiving a packet first sends it to the cluster head in its son clusters. It then sends the packet to one of the non cluster head members of its cluster which is responsible for forwarding the packet to the $m-2$ remaining members of the cluster. Figure 6 illustrate the forwarding structure along one of the cluster trees of PrefixStream. The list algorithm introduced with the cluster tree architecture can be used to balance the load among the non-head members of a cluster. The packet number i is sent to the member number i modulo $m-2$ in the list of non-head members.

Incentive PrefixStream: A node already has incentive to cooperate among its cluster otherwise other member could stop forwarding it packets. Giving incentive between clusters require bi-directional exchanges. For that purpose, we propose to use both left and right trees. Each cluster defined by prefix $x_1\cdots x_p$ is an interior cluster for the left tree rooted at $x_1\cdots x_1$ and for the right tree rooted at $x_p\cdots x_p$. In order to reduce the load of cluster heads, we propose to use two heads per cluster: one for left trees and one for right trees. If a cluster head of a parent cluster detects that a son cluster is not sending packets, it can select a different cluster head for sending packets to the son cluster until it finds a cooperative contact.

Root scheduling: The source uses a $2d$ cyclic sequence to reach root cluster heads. To allow a faster detection of cluster head failure, the source should alternate left and right trees. A possible choice thus consists in sending the first two packets to $0\cdots0$ for a left tree multicasting and then a right tree multicasting. The third and forth packets can be sent to $1\cdots1$, and so on.

Delay and load analysis: Let us first analyze the performances of this forwarding scheme. Each cluster head forwards d packets over $2d$. Once over $2d$ it acts as an interior tree node and sends the packet to $d+1$ nodes (one per son cluster plus one to a non-head

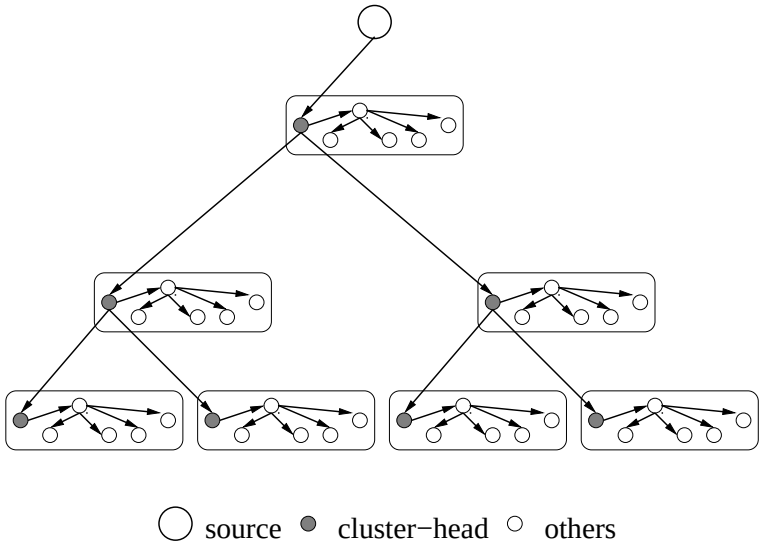

Figure 6: Multicasting inside a cluster tree of PrefixStream.

member of its cluster). $d-1$ times over $2d$, it acts as a leaf and just forwards the packet to one non-head member of its cluster. The load of a cluster head is thus $\frac{d+1}{2d} + \frac{d-1}{2d} = 1$. Non-head members of a cluster forward one packet over $m-2$ to $m-2$ other members of the cluster resulting in a load of 1. This scheme is thus ideally balanced.

A Cluster may be reduced to the two cluster heads. Each of them can then forward packets to the other without increasing its load. A cluster may also be reduced to only one cluster head which participates in both left and right trees with a load of $\frac{d+d}{2d} = 1$. The algorithm thus behaves well for any cluster size in $[1, m]$. This is an important feature because a cluster may become very small due to node failures. However, as long as it remains one alive node in the cluster the algorithm still achieves the same performances with an ideally balanced load. This gives time for the topology maintenance algorithm to reduce the prefix length defining the clusters or to make necessary ID re-assignment to re-equilibrate the clusters.

The tree height is the prefix length (in digits) minus one and thus equals $h = \log_d(n/m)$. The maximal delay is thus bounded by the tree delay $h(dT + L)$ plus the intra-cluster delay $T + L + (m-2)T + L = (m-1)T + 2L$. The intra-cluster forwarding algorithm imposes a reordering buffering window of $m-1$ packets. Figure 7 illustrates the normalized delay bounds obtained by PrefixStream (a scheme maintaining cluster size $m \in [2, 4d]$ is assumed.). We see that choosing $d = 4$ allows to obtain delays within a factor of approximately 2 from optimal for a wide range of $l = L/T$ values.

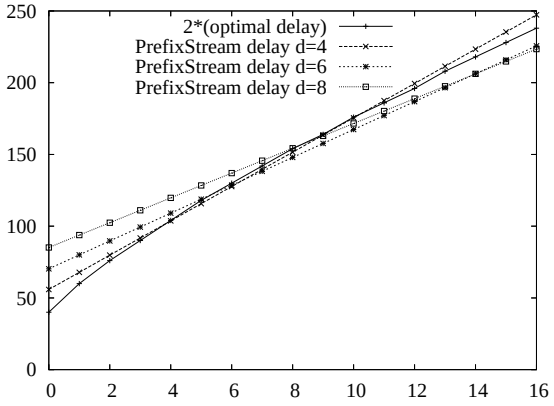

Figure 7: PrefixStream normalized delay bound as a function of $l = L/T$ for different values of d.

Network latencies optimization: The procedure for cluster head selection allows the latency of the next forwarder in the tree to be optimized by the cluster-heads. The basic selection algorithm for cluster head x consists in choosing inside a son cluster the node y with smallest round trip time $RTT = T_x + L_{xy} + T_y + L_{xy}$. To allow this, a cluster head needs to ping from time to time the members of its cluster sons. This can be done during topology maintenance exchanges. If nodes can publish their T_y value (through an estimation of their bandwidth), then x should rather select the node y with smallest $dT_y + L_{xy}$ to directly optimize the forwarding time of y along the tree. It is not clear how many nodes per cluster are required to perform a valuable optimization. However, if we consider a world wide application a node will probably find a node on the same continent in most son clusters.

Notice that we do not try to optimize intra-cluster latencies. The first reason is that the main term of delays is due to inter-cluster forwarding which account for hL when intra-cluster forwarding accounts for $2L$. It is thus more important to optimize inter-cluster latencies. The second reason is that optimizing intra-cluster latencies would result in local clusters where members would likely be in the same network. This would then be a weak point concerning reliability since a single network breakdown could disconnect several clusters and may thus break the overlay network. Forming clusters independently from the network gives better reliability with regard to network breakdowns.

Network reliability: Considering node failure, the worst case occurs when a root cluster head fails. This yields a worst failure packet loss of $1/2d$. The overall packet loss remains small. However, the packet loss inside the cluster of the failing node is $1/2$. First, an un-announced node failure inside a given cluster is a rare event compared to an un-announced node failure in the overall network. Second, to reduce further the packet loss, an

possible solution consists in using more cluster heads for leaf cluster intra-forwarding: the cluster head of each parent of a leaf cluster selects a different leaf head in the cluster. (To keep equal upload 1, this requires at least $d+1$ nodes per cluster when using the same $d-1$ leaf heads for left and right trees.) This solution still allows to optimize tree-delays except for the last hop.

Reconstruction time is very short: when a cluster head fails, the d nodes in parent clusters serving this cluster head have to be informed, thus requiring only d messages. Again, other members of the cluster may be informed later on when they have to forward packets inside the cluster.

When the number of nodes decreases, the prefix length p defining clusters may have to be reduced. To avoid a burst of topology maintenance in that case, nodes should continuously maintain topology for prefix length $p-1$. Notice that this topology contains the topology defined for prefix length p (it is d times bigger). Similarly, when the prefix length p increases, no topology maintenance is required since the new clusters are just sub-clusters of existing clusters. The only maintenance required is the possible election of new cluster heads which can simply result from sending them the packets they should forward when the moment arrives. Interestingly, the prefix length used could be decided by the source and piggy-backed in each packet. This would allow smooth transition when changing prefix length: some packets would still be forwarded according to the oldest topology while new ones would use new clusters.

Notice that a network breakdown could break the overlay topology only when a super-cluster of dm nodes disappears. If a node is affected by the breakdown with a probability p this may occur only with probability p^{dm}.

4 Conclusion

To summarize, this paper promotes three principles for achieving efficient and reliable peer-to-peer streaming: first, use interior-node-disjoint trees following the track opened by SplitStream [4] for balancing traffic loads. Second, rely on group of nodes rather than singles nodes to achieve better resilience to failure following the ideas of Nice [3], Zigzag [20] or Kademlia [14]. Third, use reciprocity to encourage cooperation between nodes in the spirit of BitTorrent [5].

To design an algorithm based on these three principles, we had to design a consistent way of defining clusters with respect to interior-node-disjoint tree construction. Interestingly, the choice of defining clusters by ID prefixes has naturally resulted in the use of the de Bruijn topology for constructing trees. One may wonder how different clustering schemes could result in different topologies.

References

[1] I. Abraham, B. Awerbuch, Y. Azar, Y. Bartal, D. Malkhi, and E. Pavlov. A generic scheme for building overlay networks in adversarial scenarios. In *Proc. of the 17th Int. Symp. on Parallel and Distributed Processing (IPDPS)*, 2003.

[2] E. Adar and B. Huberman. Free riding on gnutella. *First Monday*, 5(10), 2000.

[3] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proc. of ACM SIGCOMM*, 2002.

[4] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP)*, 2003.

[5] B. Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. of the 4th ACM SIGPLAN Symp. on Principles and practice of parallel programming*, pages 1–12, 1993.

[7] P. Fraigniaud and P. Gauron. An overview of the content-addressable network d2b. In *Brief annoucement at 22nd ACM Symp. on Principles of Distributed Computing (PODC)*, 2003.

[8] Anh-Tuan Gai and Laurent Viennot. Broose: a practical distributed hashtable based on the de-brujin topology. In *Proc. of the 4th IEEE Int. Conf. on Peer-to-Peer Computing (P2P)*, 2004.

[9] Vivek K. Goyal. Multiple description coding: Compression meets the network. *IEEE Signal Processing Magazine*, 18(5), 2001.

[10] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, and J. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation (OSDI)*, 2000.

[11] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[12] R. Karp, A. Sahay, E. Santos, and K. Schauser. Optimal broadcast and summation in the logp model. In *Proc. of ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 142–153, 1993.

[13] Gurmeet Singh Manku. A randomized id selection algorithm for peer-to-peer networks. In *Proc. of 23rd ACM Symp. on Principles of Distributed Computing (PODC)*, 2004.

[14] P. Maymounkov and D. Mazieres. Kademlia: A peerto -peer information system based on the xor metric. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[15] M. Naor and U. Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *Proc. of the 55th annual ACM symposium on Parallel algorithms and architectures (SPAA)*, 2003.

[16] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. In *ACM/IEEE NOSSDAV*, 2002.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[18] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking 2002*, 2002.

[19] D. Tran, K. Hua, and T. Do. A peer-to-peer architecture for media streaming. *IEEE JSAC Special Issue on Advances in Overlay Networks*, 2003.

[20] D. Tran, K. Hua, and T. Do. Zigzag: An efficient peer-to-peer scheme for media streaming. In *Proc. of IEEE INFOCOM*, 2003.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399